DOI 10.12694/scpe.v17i2.1159
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 17, Number 2, pp. 103–114. http://www.scpe.org
© 2016 SCPE

CLOUDFLOW - ENABLING FASTER BIOMEDICAL PIPELINES WITH MAPREDUCE AND SPARK

LUKAS FORER [§,*] ENIS AFGAN[†] HANSI WEISSENSTEINER[*§] DAVOR DAVIDOVIC[‡] GUENTHER SPECHT[§]
FLORIAN KRONENBERG[*] AND SEBASTIAN SCHOENHERR[*]

Abstract. For many years Apache Hadoop has been used as a synonym for processing data in the MapReduce fashion. However, due to the complexity of developing MapReduce applications, adoption of this paradigm in genetics has been limited. To alleviate some of the issues, we have previously developed Cloudflow - a high-level pipeline framework that allows users to create sophisticated biomedical pipelines using predefined code blocks while the framework automatically translates those into the MapReduce execution model. With the introduction of the YARN resource management layer, new computational processing models such as Apache Spark are now plugable into the Hadoop ecosystem. In this paper we describe the extension of Cloudflow to support Apache Spark without any adaptions to already implemented pipelines. The described performance evaluation demonstrates that Spark can bring an additional boost for analysing next generation sequencing (NGS) data to the field of genetics. The Cloudflow framework is open source and freely available at https://github.com/genepi/cloudflow.

Key words: Apache YARN, Pipeline Framework, Spark, Cloud Computing

AMS subject classifications. 68M14

1. Introduction. Since the advent of high-throughput technologies in the field of molecular biology (i.e. Next Generation Sequencing (NGS)), a growing amount of data is produced and needs to be analysed. Thus, molecular biology has evolved into a big data science, where the bottleneck is no longer the production of raw data in the laboratory, but its subsequent analysis and interpretation. Due to the variety of data, users need to carefully select the suitable processing framework that fits their data structure and processing task best [8]; further, the size of the data makes analysis parallelization desirable. Fortunately, a large number of conceptual approaches exist on how to deal with the data boost [14]. One promising approach for efficient data parallelization is Apache Hadoop with its YARN (Yet Another Resource Negotiator) architecture [15]. Within Hadoop, users can focus on the functional parallelization of their problem while benefiting from the scalable Hadoop architecture stack in the background. However, writing Apache Hadoop applications requires custom code development and domain expertise, which has led to poor adoption of this parallelization model in biomedical research. More specifically, the MapReduce model is quite restrictive requiring users to break an existing workflow into a number of *map* and *reduce* steps, often a challenging task. Additionally, the reusability of the map and reduce functions are limited, resulting in a use-case specific implementation and therefore time-intensive solution for every problem. To alleviate some of the pressing issues, we developed Cloudflow [6] - a framework that simplifies pipeline creation in biomedical research, especially in the field of genetics. Cloudflow supports a variety of NGS data formats and contains a rich collection of built-in operations for analyzing such kind of datasets (e.g. quality checks, mapping reads or variation calling). The main concept behind our approach is to break complex data analysis steps into three basic operations. All further use-case specific operations are built by implementing or extending one of the basic operations. Pipelines are then composed by creating a sequence of these operations. The framework itself translates the set of pipeline operations into one or more jobs and decides which of the operations are executed in the map or the reduce phase. Thus, Cloudflow hides the complexity and the implementation details of MapReduce jobs, allowing scientists to build pipelines with an intuitive method.

Initially, Cloudflow provided a framework to reuse existing Hadoop blocks for MapReduce. With the rise of the previously mentioned Apache Hadoop resource manager YARN [15], new non-batch processing models such as Apache Spark can also be run within a Hadoop cluster. The success of Apache Spark can be seen within new projects such as Adam [9], VariantSpark [11], SparkSeq [16] and especially the ongoing initiative on porting

[*]Division of Genetic Epidemiology, Medical University of Innsbruck, Innsbruck, Austria (lukas.forer@i-med.ac.at)
[†]Department of Biology, Johns Hopkins University, Baltimore, MD, USA
[‡]Center for Informatics and Computing, Ruder Boskovic Institute, Zagreb, Croatia
[§]Institute of Computer Science, Research Group Databases and Information Systems, Innsbruck, Austria

GATK to Spark (GATK 4). However, similar to MapReduce, writing Spark pipelines can be a challenging task that prevents domain experts from using such models in their daily work. In the future, it is likely that Apache Spark will substitute MapReduce as the general-purpose processing engine of Hadoop. To support this transition, we extended Cloudflow to support Spark and evaluated three genetics data-analysis pipelines that have been developed on both MapReduce and Spark engines. The results demonstrates that our contribution helps reduce the development time and increases reusability of the code over different use cases and even on different data processing engines.

2. Apache Hadoop. Apache Hadoop is an initiative for distributed computing on a cluster infrastructure. It is well known for providing an open-source implementation of the batch processing model MapReduce, initially developed by Google in 2004. In 2013, Apache Hadoop introduced a new resource management system - YARN - that allows multiple data processing models to be integrated on the same cluster, allowing different processing models to share common underlying resources. In YARN, the MapReduce processing library is now referred to as MapReduce 2 (MR2). MR2 uses the required distributed file system (HDFS) as the default data location for data input and output.

Compared to MapReduce 1 (MR1), the two main components (JobTracker, TaskTracker) have been reorganized: The *JobTracker* has been broken up into three services (ResourceManager, Application Master, JobHistoryServer). The *ResourceManager* is a YARN service to run applications (e.g. MapReduce, Impala or Spark). The *Application Master* is started for each job and includes all parts to manage e.g. a MapReduce job. The *JobHistoryServer* is responsible for providing information about already finished jobs. Instead of using a *TaskTracker* to start individual tasks, the YARN *NodeManager* service manages all resources on one node and starts individual *containers* (former tasks). As mentioned, this setup provides multiple processing models within the same cluster, including MapReduce and Spark.

One of the important changes within YARN is how resources are managed. Within MapReduce, the amount of concurrent tasks on each node have been specified separately for map and reduce (*mapred.tasktracker.map.tasks.maximum*, *mapred.tasktracker.reduce.tasks.maximum*). Now with YARN, one can only specify the amount of cores on each node (*yarn.nodemanager.resource.cpu-vcores*) and the amount of available memory (*yarn.nodemanager.resource.memory-mb*), independently of map and reduce. Using these two parameters, YARN runs the amount of appropriate containers per node. Since Cloudflow provides support for both the MapReduce and Spark processing model, they will be introduced shortly in the coming paragraphs.

2.1. MapReduce. The aim of MapReduce is to develop a simple and scalable method to process large datasets on several machines in parallel [5]. The main idea behind this distributed programming model is that a long-time calculation is split into a *map* and a *reduce* phase which contains all the logic behind the calculation and is specified by the user. The underlying framework itself takes care of parallelization, task scheduling, load-balancing and fault-tolerance. Due to its simplicity, MapReduce is used to solve many scientific problems where large-scale computing is needed. Moreover, MapReduce is ideal for parallel batch processing of terabytes of input data. The data-flow of a MapReduce program consists of several steps, where only the map and reduce function are problem-specific and the other steps are loosely coupled with the problem and generalized. In the first step, the input data set is split into key/value pairs and a user defined map function is executed for each pair:

$$map(key, value) \rightarrow list((key_i, value_i)), where_i = 0...n$$

The map function reads a pair, performs some problem-specific calculations and produces zero or n intermediate key/value pairs for each input pair. In the next step, the intermediate key/value pairs are grouped by similar keys and a merged list of all values for this key is created. Finally, the user-defined reduce function is applied to the intermediate key/list pairs:

$$reduce(key_i, list(value_i)) \rightarrow list(outvalue)$$

The values created by the reduce function are the final outputs of a MapReduce job.

2.2. Apache Spark. The cluster computing framework Spark was initiated at UC Berkeley AMPlab in 2009 and is an Apache top-level project since 2014 [17]. Spark can run in standalone mode, or over existing cluster managers like Apache Mesos or YARN. Therefore, no Hadoop cluster is required as long as a shared file system is provided (e.g. NFS). In addition to the batch data processing model, streaming data processing model is supported by using a micro-batch model. Currently Spark offers four main libraries: Spark SQL and Dataframes for working with structured data, Spark Streaming for fault-tolerant stream processing, a machine learning library called MLlib, with a collection of algorithms already provided and an API for graph processing called GraphX.

Spark's main differentiation to MapReduce is the in-memory cache based on the Resilent Distributed Datasets (RDD) concept. Thereby it has lower launching overheads compared to MapReduce [16]. While MapReduce is ideal for batch processing of large datasets, Spark profits from RDDs caching and is therefore able to handle tight-coupled problems faster than MapReduce. If the data fits in the memory, Spark outperforms MapReduce. In contrast to the two-stage disk based Map and Reduce phase, Spark allows multi-stage in-memory primitives, based on RDDs transformation and action operations. While *map* can be modeled with a lazy transformation that calls a function for each element in a data set and returns another RDD, *reduce* is an action operation triggering a computation (aggregation) and sending either all the RDDs elements back to the main program (*driver*) or write them on the disk. The RDD's operation concept is represented in Figure 2.1.

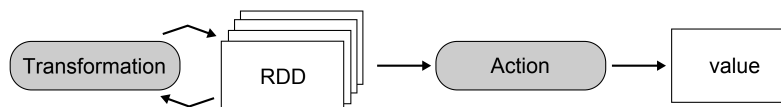

Fig. 2.1. *Apache Spark's RDD Operations.*

Spark based software in the genomic data analysis is currently evolving, by exploiting the increased memory of cluster systems. First implementations of the Smith-Waterman local sequence alignment approach was realized within SparkSW [18]. VariantSpark [11] employs Spark MLlib machine learning library using the k-means clustering for population structure deconvolution.

2.3. Related Work. To simplify the implementation of MapReduce jobs, Apache Pig has been introduced as a higher level interface [12]. It is based on HDFS and MapReduce and allows a fast implementation of data flows with already available data operations such as *join*, *filter* or *group by.* Data flows are specified in the Pig Latin language that describes a directed acyclic graph (DAG) and defines how data should be processed. A further advantage of Apache Pig is the ability to check the data flow on optimizations, e.g. if two grouping statements can be combined. The costs to write code in Apache Pig are lower than setting up a Java project and implementing the corresponding MapReduce functions. But, as stated earlier, Apache Pig includes only a limited number of operators and writing Apache MapReduce jobs directly in Java yields advantages in speed compared to Apache Pig.

The pipeline framework FlumeJava [4] is based on the concept of immutable parallel collections. This kind of data-structure can be used to process and analyze their items in parallel. The end user can either use one of the predefined functions or can combine them with its own. Each function is implemented as parallel for-each loop, which is then translated by the framework into a series of MapReduce jobs. During the translation, the framework itself decides if such a function should be executed locally (i.e. sequentially) or remotely (i.e. parallel). Apache Crunch (https://crunch.apache.org/) is a freely available open source implementation of FlumeJava.

3. Cloudflow. Cloudflow [6] is a framework to simplify the creation of analysis pipelines by encapsulating complex data analysis steps as simple operations. This approach helps hide the complexity and the implementation details of complex data-parallel pipelines. Moreover, the concept of using basic operations increases the reusability and enables testing of the operation logic on a local workstation by using existing unit testing frameworks.

Since Cloudflow used initially Apache MapReduce for pipeline execution, it offers parallel data processing, data reliability and fault tolerance out of the box. This fact is especially important in the field of Cloud Computing, where infrastructure often relies on commodity hardware and nodes can fail on a regular basis (e.g. due

to miss-configuration or hardware failures). At the same time, the architecture of Cloudflow is independent from MapReduce. As we show in this paper on the example of Apache Spark, it provides parallelization constructs and abstraction interfaces that can be used to extend the system by implementing other parallel programming models (see Figure 3.1).

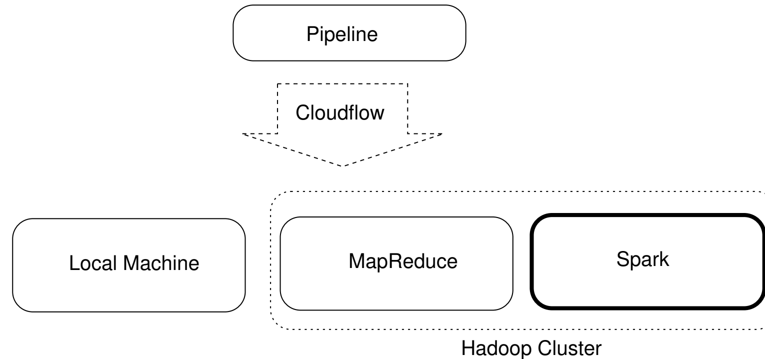

Fig. 3.1. *The architecture of Cloudflow enables to test the pipeline during development on the local machine and to execute the same pipeline on a Hadoop cluster for big data analaysis.*

Instead of developing a new declarative language for the pipeline composition, we developed a clear Java API. The proposed framework implements different patterns to speed up pipeline creation, to be extensible, and to support test-driven pipeline development. The following section gives an overview on the abstraction, explains the basic operations in detail, and shows how pipelines are created.

3.1. Data Types and Basic Operations. Cloudflow operates on records consisting of a key/value pair, whereby different record types are available (e.g. *TextRecord*, *IntegerRecord*, *FastqRecord*). A loader class is responsible for loading the input data and converting it into an appropriate record type. As mentioned earlier, Cloudflow supports three different basic operations: *transform*, *summarize*, and *group*. These are used to analyze and transform records.

The **transform operation** is used to analyze one input record and to create between 0 and n output records. The user implements the computational logic for this operation by extending an abstract class. This child class provides the function implementation that is then executed by the Cloudflow framework for all input records in parallel:

```
Class MyTransformer extends Transformer {
    public void transform(Record) {
       doSomethingInParallel();
       emit(new Record());
    }
}
```

The **summarize operation** is used on a list of records, whereby records with the same key are grouped. Thus, the signature of the process method has the key and a list of records as an input:

```
Class MySummarizer extends Summarizer {
   public void summarize(Key, List<Record>) {
       doSomethingInParallel();
       emit(new Record());
    }
}
```

The **group operation** is a special operation, which takes a list of records as an input and creates a Record group with the same key. The Cloudflow framework automatically inserts a group-operation between a transform- and a summarize-operation. This ensures, that output records of the transform operation are compatible with the input records of the summarize operation. The group-operation is realized by using the shuffle phase of a MapReduce job or *reduceByKey* transformation in Spark.

Based on these operations, the user defines pipelines by building sequences of operations (see Section 3.3); a pipeline must start with a transform operation, all further operations are optional and can be used in arbitrary order.

3.2. Extended Operations. Complex operations are built by combining one or more basic operations. We already implemented several standard operations that are helpful for the analysis of text or numerical data records:

- Filter: this operation is a special transform operation, which emits the record to the subsequent operation iff a user-defined condition is fulfilled.
- Split: the transform operation calculates for each input record a new split level (i.e. a new key). This key is used by the group operation to create chunks, which can then be analyzed by a user-defined summarize operation.
- Aggregation (sum, mean): This defines a group operation followed by a summarize operation to aggregate all values with the same key (e.g. calculates the mean of all values). One record with the aggregated value is then emitted to the subsequent operation.
- Executor: this summarize operation writes all grouped values into a file on the local disk. It is then used as the input of an external UNIX command line program. Based on the lines of the output file, new records are created and emitted to the subsequent operation.

Since Cloudflow's operations are based on the Composite pattern, all these extended operations can also be used as a basis for new operations. In addition, this enables to split complex operations into several sub-operations, which improves testing and maintenance.

3.3. Pipeline Composition. The user builds pipelines by connecting several operations with compatible interfaces. For this purpose the Cloudflow framework implements the Builder pattern, which enables (a) building complex pipelines, (b) providing type safety and (c)support for implementing domain specific builders (see Section 4.1). In addition, the Builder pattern ensures that only a valid sequence of operations can be created (e.g. after the group-by operation, a summarize operation has to be added).

```
Class LineToWords extends Transformer {
   public void transform(TextRecord rec) {
    String[] words = rec.getValue().split()
    for (String word: words){
      emit(new IntegerRecord(word, 1));
    }
   }
}
pipeline.loadText(input)
        .transform(LineToWords.class)
        .sum()
        .save(output);
```

Fig. 3.2. *WordCount example using Cloudflow.*

To help the user and accelerate the pipeline composition process, Cloudflow comes preconfigured with a set of useful operations. This has the advantage that even the default WordCount example can be broken down into a few simple operations and is defined in a single line of code (see Figure 3.2). In the first step, a text file is loaded from HDFS (loadText). Then, for each record (i.e. line of input) the application-specific *LineToWords* operation is executed, which splits the line into words and creates a new record for each word. In the last step, a predefined sum operation is executed. The operation extends the pipeline by a group-by operation and a summarize-operation in order to sum up all the values for a certain key. For frequently used operations (e.g. sum, mean or count), we created special builder functions, which extend the pipeline and improve the code readability by keeping the code simple.

3.4. Pipeline Execution on MapReduce. Before the execution, Cloudflow checks the compatibility of input and output records of consecutive operations. This ensures that only valid and executable pipelines are submitted to the cluster.

If the pipeline is executable and valid, then the operation sequence is translated into an execution plan. The execution plan specifies if an operation is executed in the map or in the reduce phase. Based on this plan, Cloudflow creates one or more MapReduce jobs and configures them to execute the user-defined operations in the correct order. In this translation step, Cloudflow tries to minimize the number of MapReduce jobs by combining consecutive transform operations and by executing all transform operations after a summarize operation in the same reducer instance (see Figure 3.3).

For additive summarize operations (e.g. sum), Cloudflow takes advantage of Hadoop's combiner functionality. The idea of this improvement is to combine the key/value pairs that are generated by all the map tasks on the same machine into fewer pairs. Thus, the number of pairs that are transferred between mapper and reducer are minimized, which results in a positive effect on the network bandwidth since unnecessary communication is avoided.

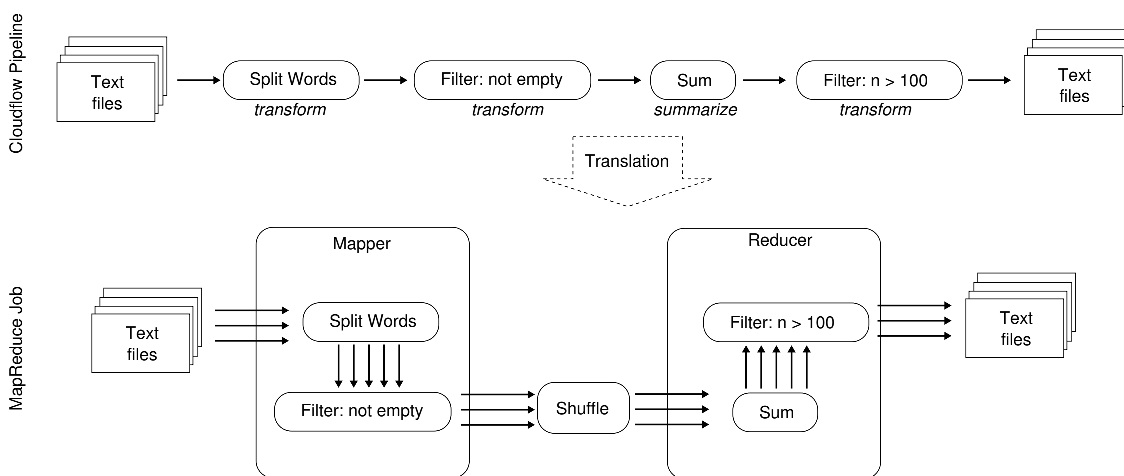

Fig. 3.3. *Cloudflow translates the operation sequence automatically into an executable MapReduce job.*

3.5. Pipeline Execution on Spark. Since the architecture of Cloudflow is independent from MapReduce, the provided parallelization constructs and abstraction interfaces can be used to extend the system by using Spark as an alternative parallelization framework. Thus, we implemented a new pipeline executor that takes a Cloudflow pipeline as input and translates the execution plan to a sequence of transformations. This transformations are using the Spark API in order to operate on Resilient Distributed Datasets (RDD) and take advantage of in-memory technologies.

The main task of this process is to combine all consecutive transform operations and to execute them step by step in the *flatMapToPair* stage of a Spark job. Next, the summarize operation defines the logic behind the *groupByKey* operation. In the last step, all transform operations are then executed using Spark's *map* transformation. Thus, the created Spark job needs two stages to execute a single Cloudflow pipeline (see Figure 3.4).

To optimize the execution time of the generated Spark job, Cloudflow tries to minimize the number of memory intensive *groupByKey* operations. This is achieved by using *reduceByKey* for most summarize-operations or by mapping this operations to the equivalent build-in transformation of Spark (e.g. count, sum, mean).

3.6. Bioinformatics Support. Cloudflow provides a variety of already implemented utilities, which facilitate the creation of pipelines in the field of Bioinformatics (especially for NGS data in Genetics). For that purpose, we implemented, based on HadoopBAM [10], several record types and loader classes in order to process FASTQ, BAM and VCF files. Moreover, we created several operations and filters for the analysis of biological

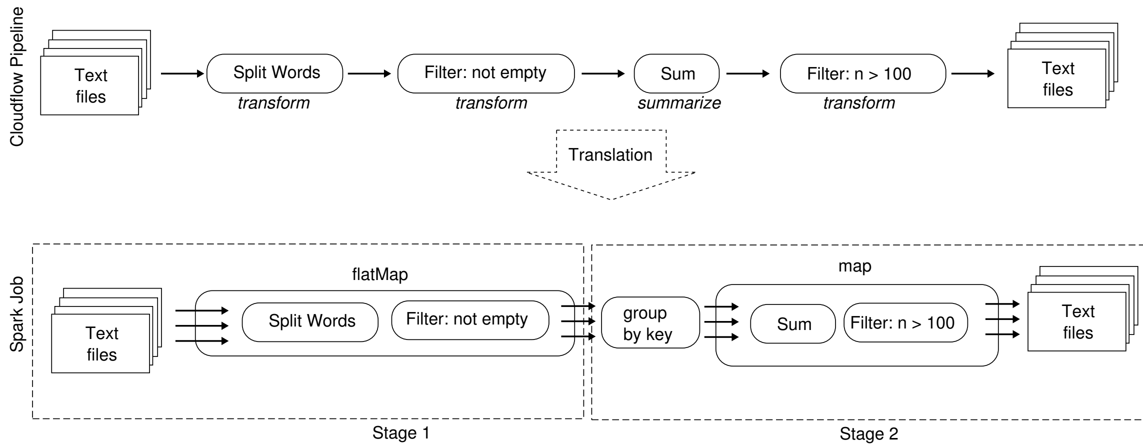

Fig. 3.4. *The Spark pipeline executor takes a Cloudflow pipeline as input and translates the execution plan to a sequence of Spark transformations.*

datasets. For example, a typical quality control pipeline for VCF files can be implemented by simply combining several built-in operations. First, we apply predefined filters to discard variations that are monomorphic, marked as duplicates, or are Insertions or Deletions (InDels). For all records passing the filters, Cloudflow applies a summarize operation that calculates the call rate for each variation (see Figure 3.5).

```
Class CallRateCalc extends Transformer {
  public void transform(VcfRecord record) {
    VariantContext snp = record.getValue();
    float call = callRate(snp);
    emit(new FloatRecord(snp.getID(), call);
  }
}

pipeline.loadVCF(input)
        .filter(MonomorphicFilter.class)
        .filter(DuplicateFilter.class)
        .filter(InDelFilter.class)
        .transform(CallRateCalc.class)
        .save(output);
```

Fig. 3.5. *VCF Quality Control Pipeline using Cloudflow.*

3.7. Deploying Pipelines as a Service. Cloudgene [13] is a web-based platform to create and execute workflows consisting of Hadoop YARN, Apache Pig and command line-based programs. It can be seen as an additional layer between Hadoop and the end-user that hides the complexity of the framework. Therefore, Cloudgene is the perfect candidate to provide Cloudflow pipelines as a service. Such pipeline can be integrated into the workflow platform by utilizing Cloudgene's plugin interface. No adaptation to the source code is needed, while only a simple plain text file including a header, input parameters, output parameters and the definition of the workflow itself need to be created. When launching Cloudgene, the manifest file is loaded and the client interface is automatically rendered using information from the file. As Cloudgene supports different technologies, it is possible to parallelize the calculations using Cloudflow and to visualize the results using R.

Cloudflow requires a compatible Hadoop YARN cluster for executing pipelines. CloudMan [2, 3] makes it possible to easily procure and configure a virtual compute cluster on a cloud infrastructure. The procured platform delivers a dynamically scalable Slurm and Hadoop cluster along with a number of higher level bioin-

formatics applications. With its ability to be launched and managed via a web browser on a number of clouds, customized as necessary, and easily shared with collaborators, CloudMan makes it possible to readily utilize cloud resources in a research environment. The approach on how Cloudgene and CloudMan can be combined efficiently has already been demonstrated [7], paving the path for readily executing Cloudflow pipelines.

4. Evaluation. As shown in [6], Cloudflow has practically the same execution time compared to Apache Crunch and the overhead between a Cloudflow pipeline and a plain MapReduce job is negligible. To evaluate the here presented extension, we implemented three different data-analysis pipelines using Cloudflow. The results of our experiments demonstrate that Spark brings a speed-up in the execution time compared to MapReduce. The following sections describe the experiments in detail. For more extensive pipeline examples, take a look at the source code repository at https://github.com/genepi/cloudflow. All the experiments have been executed on a 6 node physical cluster (5 cores & 32 GB RAM each) running the latest Cloudera Version (CDH 5.1.1) and Hadoop YARN 2.6.

4.1. WordCount. We extended our previously introduced WordCount example from [6] and executed it with different input datasets to compare the execution times of the existing Cloudflow MapReduce implementation with the new Spark implementation (see Figure 4.1).

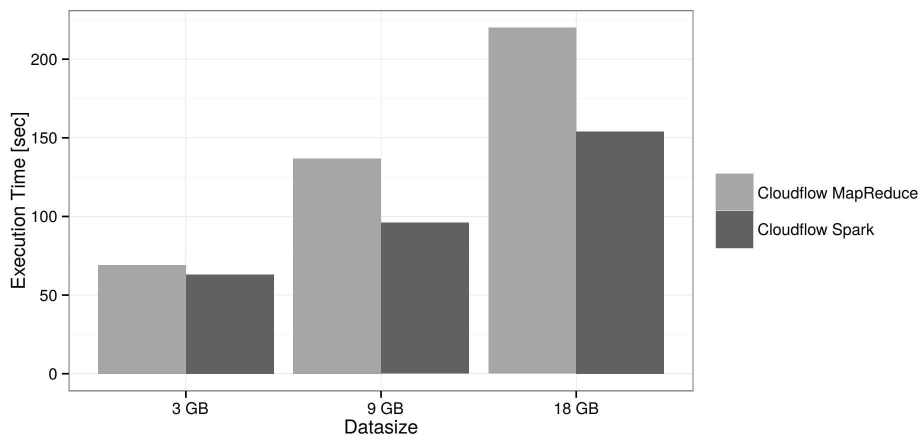

Fig. 4.1. *Execution time of the WordCount use case*

The results of this initial experiment show that overall the new Spark executor is faster than the existing MapReduce implementation. Moreover, the results demonstrate that the Spark WordCount pipeline implemented within Cloudflow scales well with the amount of input data.

4.2. Quality Control of Sequencing Data. When working with NGS data, the quality of raw data needs to be checked before subsequent downstream analysis (e.g. read mapping/alignment) can be achieved. Factors such as read errors, insertions or deletions of bases must be considered that finally results in the most accurate genome position for each read. The mean base quality per position is an important metric to get an overview about sequencing data quality.

The parallelization of the pipeline is realized by writing a transform operation, which calculates a quality value for each base of the input read. Next, a summarize operation is used to calculate the mean value of all qualities for a certain position (see Figure 4.2).

We executed the pipeline with different BAM files taken from the 1000 Genomes Project [1] to test and validate the pipeline with real-world input data. Three different datasets were used: (1) sample HG00097 high coverage chromosome 11 [1 GB]; (2) sample HG00096 high coverage whole exome [8 GB] and (3) sample HG00096 low coverage whole genome [15 GB]. The results of this experiment show that the new Spark executor has a speed-up of 2 compared to the MapReduce implementation (see Figure 4.3).

```
class GetQuality extends Transformer<BamRecord, IntegerRecord> {
    public void transform(BamRecord record) {
    for (int pos = 0; pos < (record.getSequenceLength(); pos++) {
        emit(new IntegerRecord(pos, (record.getQualityAtPos(pos));
    }
}
BioPipeline pipeline = new BioPipeline("Check Base Quality");
pipeline.loadBam(input).apply(GetQuality.class).mean().save(output);
```

FIG. 4.2. *BAM Quality Control Pipeline using Cloudflow.*

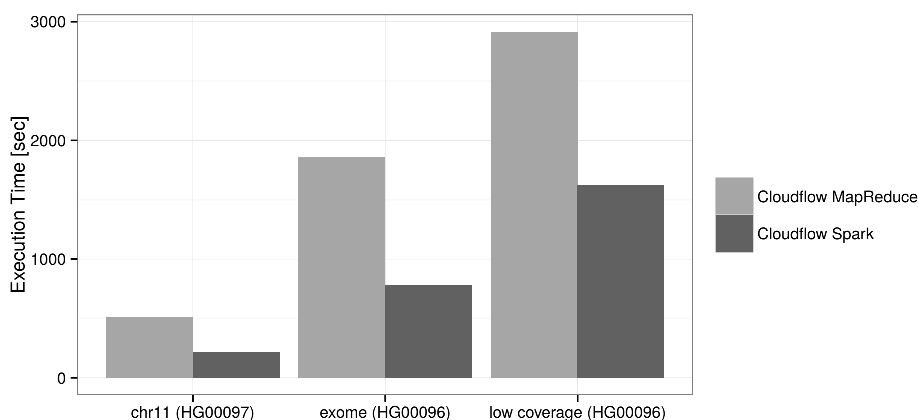

FIG. 4.3. *Execution time of the BAM Quality Control Pipeline with different sample from the 1000 Genomes Project*

For this pipeline Cloudflow uses the *groupByKey* transformation to calculate the average of all qualities. This transformation shows especially a good performance if all values for one key can fit into memory. However, too many values resulting in disc swaps and finally resulting in a execution time similar to MapReduce. This could lead to a bottleneck for extremely large datasets.

4.3. Sequence Length Distribution. This workflow generates the distribution of all sequence lengths within one BAM sample. This is especially useful for currently upcoming technologies such as the third generation Single Molecule Sequencing on PacBio or Oxford Nanopore systems to check for long reads or to determine appropriate reads as seeds within mapping algorithms.

The parallelization of the pipeline is realized by writing a transform operation that creates a new record for each read where the key represents the length of the read and the value is set to a constant. The summarize operation sums up all values per key, which finally represents the total number of reads with a certain length (see Figure 4.4).

```
class CalcReadLength extends Transformer<BamRecord, IntegerRecord> {
    public void transform(BamRecord record) {
        emit(new IntegerRecord((record.getSequenceLength(), 1);
    }
}
BioPipeline pipeline = new BioPipeline("Check Read Length");
pipeline.loadBam(input).apply(CalcReadLength.class).sum().save(output);
```

FIG. 4.4. *Sequence Length Distribution using Cloudflow.*

The pipeline was executed with the same datasets as described in Section 4.2. Again this experiment shows

the expected speed-up when the Cloudflow pipeline is executed on Spark instead of MapReduce (see Figure 4.5).

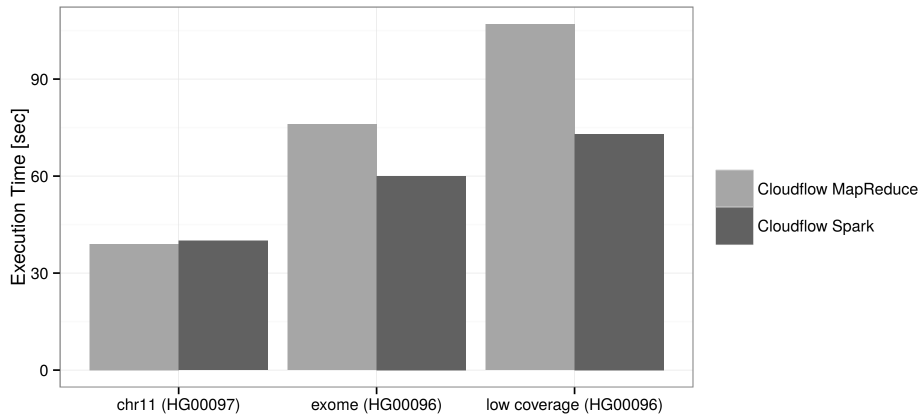

Fig. 4.5. *Execution time of the Sequence Length Distribution pipeline with different samples from the 1000 Genomes Project*

Compared to the previous pipeline, Cloudflow uses the *reduceByKey* transformation to sum up all values. This enables Spark to combine records with the same key on each partition before shuffling the data and sending it to other nodes. Thus, no memory overflow is produced and the pipeline scales linear with the amount of data.

4.4. Scalability. We tested the scalability of the new Spark executor in terms of the number of machines. For that experiment, we executed the same Cloudflow pipeline on the Hadoop cluster using 2, 4 and 6 nodes and compared the measured execution times with those of the MapReduce executor (see Figure 4.6). The results show that the overhead of executing a Cloudflow pipeline on a small MapReduce cluster (in our case two nodes) is much higher than using the same cluster with Spark. However, after a cluster size of 4 nodes both executors scale identically. Thus, the user benefits from the Spark executor when the used cluster is small.

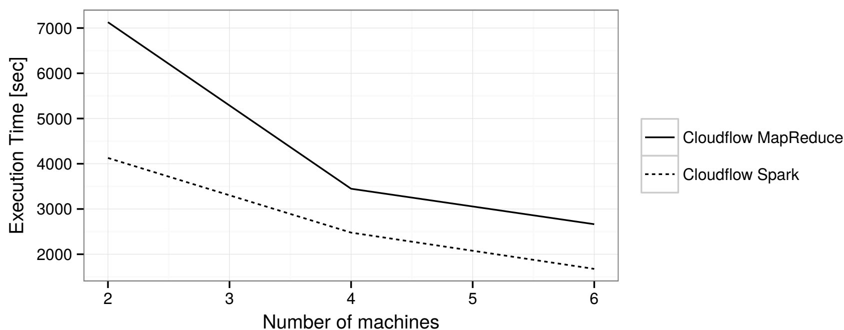

Fig. 4.6. *Execution time of the WordCount use case*

5. Discussion and Future Work. With Coudflow, an approach was presented that enables separation of the development of analysis pipelines from the underlying parallelization model. The extension of Cloudflow to Spark as an alternative execution engine has several advantages: (1) the developer has the possibility to create pipelines without thinking about the underlying technical details of the used parallelization framework; (2) the developer is not limited to one specific parallelization framework and can evaluate its pipeline using different frameworks without additional effort; (3) the end-user has the possibility to decide where an existing pipeline should be run depending on its computing infrastructure. Moreover, users take advantage when new

parallelization models are integrated into Cloudflow and can use them without changing the source code of the pipeline.

However, such a generic approach also has some limitations in terms of optimizations. Currently, Cloudflow does not take advantage of all in-memory features provided by Spark. Our implementation minimizes the number of memory intensive transformations (e.g. *groupByKey*) only for some predefined 'summarizers' (e.g. sum and count). Moreover, no caching-support is implemented, which is essential for machine learning algorithms. To overcome this issue we will implement additional features in future releases to support even more use cases in the field of bioinformatics. Beside the optimizations of the translation processes, we plan to integrate new file formats such as ADAM and CRAM, which will enable further development of scalable services.

6. Conclusion. Cloudflow's overall aim is to simplify the development of complex analysis pipelines by using abstract operations. Therefore, operations need only be written once and can be re-used for future MapReduce and Spark usage. The major advantage of Cloudflow lies in the provision of validated operations, especially in the area of genetics, and its extensibility.

The contribution of this paper was an extension of Cloudflow that enables executing the same pipeline on Spark as on MapReduce. The experiments and evaluation show that (1) the same pipeline can be executed on Spark without changing the existing source code, (2) Spark leads to a speed-up for most pipelines and (3) the architecture of Cloudflow is extensible and therefore new analysis tools and file-formats based on Spark (e.g. ADAM) can be easily integrated in the future.

Acknowledgments. This work was, in part, supported by the "Scalable Big Data Bioinformatics Analysis in the Cloud" grant from the Croatian Ministry of Science, Education, and Sport and the Austrian Federal Ministry of Science and Research (BMWF) and by the FP7-PEOPLE programme grant 277144 (AIS-DC).

REFERENCES

[1] G. R. Abecasis, A. Auton, L. D. Brooks, M. A. DePristo, R. M. Durbin, R. E. Handsaker, H. M. Kang, G. T. Marth, and G. A. McVean. *An integrated map of genetic variation from 1,092 human genomes.* Nature, 491:56–65, 2012 Nov 1 2012.

[2] E. Afgan, D. Baker, N. Coraor, B. Chapman, A. Nekrutenko, and J. Taylor. *Galaxy cloudman: delivering cloud compute clusters.* BMC bioinformatics, 11(Suppl 12):S4, 2010.

[3] E. Afgan, D. Baker, N. Coraor, H. Goto, I. M. Paul, K. D. Makova, A. Nekrutenko, and J. Taylor. *Harnessing cloud computing with galaxy cloud.* Nature biotechnology, 29(11):972–974, 2011.

[4] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. *Flumejava: easy, efficient data-parallel pipelines. In* ACM Sigplan Notices, volume 45, pages 363–375. ACM, 2010.

[5] J. Dean and S. Ghemawat. *Mapreduce: simplified data processing on large clusters.* Communications of the ACM, 51(1):107–113, 2008.

[6] L. Forer, E. Afgan, H. Weissensteiner, D. Davidovic, G. Specht, F. Kronenberg, and S. Schönherr. *Cloudflow - A framework for mapreduce pipeline development in biomedical research. In* 38th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2015, Opatija, Croatia, May 25-29, 2015, pages 172–177, 2015.

[7] L. Forer, T. Lipic, S. Schonherr, H. Weisensteiner, D. Davidovic, F. Kronenberg, and E. Afgan. *Delivering bioinformatics mapreduce applications in the cloud. In* Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2014 37th International Convention on, pages 373–377. IEEE, 2014.

[8] T. Lipic, K. Skala, and E. Afgan. *Deciphering big data stacks: An overview of big data tools. In* Big Data Analytics: Challenges and Opportunities (BDAC-14), 2014.

[9] M. Massie, F. Nothaft, C. Hartl, C. Kozanitis, A. Schumacher, A. D. Joseph, and D. A. Patterson. *Adam: Genomics formats and processing patterns for cloud scale computing.* Technical Report UCB/EECS-2013-207, EECS Department, University of California, Berkeley, Dec 2013.

[10] M. Niemenmaa, A. Kallio, A. Schumacher, P. Klemelä, E. Korpelainen, and K. Heljanko. *Hadoop-bam: directly manipulating next generation sequencing data in the cloud.* Bioinformatics, *28(6):876–877, 2012.*

[11] A. R. O'Brien, N. F. W. Saunders, Y. Guo, F. A. Buske, R. J. Scott, and D. C. Bauer. VariantSpark: population scale clustering of genotype information. BMC Genomics, 16(1):1052+, Dec. 2015.

[12] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. *Pig latin: a not-so-foreign language for data processing. In* Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pages 1099–1110. ACM, 2008.

[13] S. Schönherr, L. Forer, H. Weissensteiner, F. Kronenberg, G. Specht, and A. Kloss-Brandstätter. *Cloudgene: A graphical execution platform for mapreduce programs on private and public clouds.* BMC bioinformatics, 13(1):200, 2012.

[14] O. Spjuth, E. Bongcam-Rudloff, G. C. Hernández, L. Forer, M. Giovacchini, R. V. Guimera, A. Kallio, E. Korpelainen, M. M. Kańduła, M. Krachunov, et al. *Experiences with workflows for automating data-intensive bioinformatics.* Biology direct, 10(1):1–12, 2015.

[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. *Apache hadoop yarn: Yet another resource negotiator. In* Proceedings of the 4th annual Symposium on Cloud Computing, page 5. ACM, 2013.

[16] M. S. Wiewiorka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski. *SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision.* Bioinformatics, 30(18):2652–2653, Sept. 2014.

[17] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. *Spark: Cluster computing with working sets. In* Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[18] G. Zhao, C. Ling, and D. Sun. *Sparksw: scalable distributed computing system for large-scale biological sequence alignment. In* Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on, pages 845–852. IEEE, 2015.

Edited by: Karolj Skala
Received: December 21, 2015
Accepted: March 31, 2016